Data composition patterns in service-based workflows

Johan Montagnat, Tristan Glatard, Diane Lingrand

Data composition patterns in service-based workflows

Johan Montagnat, Tristan Glatard, Diane Lingrand
I3S laboratory, CNRS, Sophia Antipolis, France
http://www.i3s.unice.fr/~johan

Abstract

Workflow engines are powerful tools to implement data-intensive scientific applications exploiting parallel grid resources transparently. We discuss the advantages of implementing applications as workflows of services when dealing with large data sets. We show how the graph of services associated with data composition operators enable the description of complex data flows in a very compact format. We define a strict semantics for two common composition operators and we propose an algorithm to consistently satisfy this semantics all along the workflow execution. Finally, we show how our approach enables parallel execution of the application while preserving the data flow semantics. We implemented the algorithm proposed in MOTEUR, an open source workflow engine designed to execute parallel and data-intensive applications.

1. Introduction

Assembling basic processing components is a powerful mean to develop new scientific applications. The reusability of data processing software components considerably reduces applications development time. Workflow description languages and execution engines ease the development of such applications by enabling an abstract definition of the applications logic through a high level representation. They provide the ability to chain the application components execution while respecting causality and inter-components dependencies expressed with this abstract representation.

We are focusing on the execution of scientific workflows requiring the manipulation of large amounts of data. In this context, it is common to build a scientific data analysis procedure and to execute it on a large amount of different data segments to be analyzed. The input data is usually composed of both scientific measurements to be processed and algorithms parametrization values. It is therefore important not only to describe the workflow graph, but also the different data sets to be processed resulting in multiple data flows. The description of the input data needs to be independent from the workflow description itself to enable re-execution on different data sets and/or with different parameter values without completely rewriting the workflow.

The description of data flows is better handled through the *service-based* submission strategy as discussed in section 2. This strategy enables the definition of powerful data composition patterns (section 3) in a compact way. In this paper, we propose an algorithm for helping the user in composing data flows intuitively (section 4). We discuss the problems arising when exploiting parallelism for optimizing the use of grid resources in section 5 and we illustrate our approach through a real application to medical images analysis in section 6.

2. Handling data flows

Workflow managers can be broadly classified into two main categories: *control-centric* and *data-centric*. The control-centric managers, are more focused on the description of complex application flows. They provide an exhaustive list of control structures such as branching, conditions and loop operators. They can describe very complex control composition patterns [1, 10, 18] and some of them are comparable to small programming languages, including a graphical interface for designing the workflow and an interpreter for its execution. Conversely, data-centric managers usually provide a more limited panel of control structures and rather focus on the execution of heavy-weight algorithms designed to process large amounts of data. The complex application logic is supposed to be embedded inside the basic application components. Although there is *a priori* no contradiction in implementing a workflow manager that is both control and data-centric, the optimization of different managers for different needs often leads to the splitting between these two trends. Control-centric managers are commonly implemented to fulfill the e-business community needs. Applications are often not so compute nor data-intensive and can be described in a high level language suitable for non-experts. Conversely in the scientific area, complex application codes, both compute and data intensive, are frequently available. The workflow description

languages are not so rich but the execution engines are better taking into account execution efficiency and data transfer issues [20]. In the remaining of this paper, we will consider scientific workflow managers only.

With the arising of production grid infrastructures[4, 16, 14], data-intensive applications have emerged in many scientific areas for which workflow processing is particularly well adapted. In these applications, the workflow manager should not only enable the description and the control of the application logic, but it should also provide means of describing and controlling the data flows. Following on the main job submission and control strategies used in grid computing, two main approaches have been proposed to handle scientific workflows. We refer to them as the the *task-based* and the *service-based* approaches:

1. In the *task-based* strategy, also referred to as *global computing*, users define computing tasks to be executed. Any executable code may be requested by specifying the executable code file, input data files, and command line parameters to invoke the execution. The task-based strategy, implemented in GLOBUS [5], LCG2[1] or gLite[2] middlewares for instance, has already been used for decades in batch computing. An emblematic workflow manager using the task-based approach is the Directed Acyclic Graph Manager (DAGMan) from Condor [3]. Many other works in this framework, such as VDS [7] and Pegasus [3], are built on top of DAGMan.
2. The *service-based* strategy, also referred to as *meta computing*, consists in wrapping application codes into standard interfaces. Such services are seen as black boxes from the workflow manager for which only the invocation interface is known. Various interfaces such as Web Services [19] or gridRPC [13] have been standardized. The services paradigm has been widely adopted by middleware developers for the high level of flexibility that it offers (*e.g.* in the OGSA [6] and the WS-RF extension to web services). However, this approach is less common for application codes as it requires all codes to be instrumented with the common service interface. For instance, the globus toolkit is built on service-oriented framework. The GRAM-WS services enable user tasks submission and monitoring. Yet, the user codes executed are tasks initiated through a command line call.

The service-based approach has been adopted **for user code submission** in well-known workflow managers such as the Kepler system [11], Taverna [15], Triana [17] or MOTEUR [8].

[1] LCG2 middleware, http://lcg.web.cern.ch/LCG/activities/middleware.html

[2] gLite middleware, http://www.glite.org

[3] Condor DAGMan, http://www.cs.wisc.edu/condor/dagman/

The main difference between the task-based and the service-based approaches is the way the data sets to be processed are handled. In the task-based approach, input data are specified with each task. This representation mixes data and processing descriptions. The dependency between two tasks is explicitly stated as a data dependency in these two task descriptions. This representation is static and convenient for optimizing the corresponding computations: the full oriented graph of tasks is known when the computations are scheduled, thus enabling many optimization opportunities for the workflow scheduler [2].

Conversely, the service-based approach decouples data and processings. Input data are dynamically specified at execution time as input parameters to the workflow manager. Each service is defined independently from the data to be processed and it is only at the service invocation time that the input data is sent to the service. This eases the reexecution of application workflows on different input data sets. In this framework, the dependencies between consequent services are logically defined at the level of the workflow manager. Each service is designed independently from each other.

2.1. Data-intensive applications on grids

When considering data-intensive applications for which grids offer a proper computing support, a user often needs to define a processing workflow that will not apply to a single piece of input data, but rather to full data sets. Each piece of data in the input data set has to follow the same processing flow, independently from the other inputs.

In the task-based approach, two input data, even being processed by a same algorithm, result in the definition of two independent tasks. This becomes very tedious, and even quickly humanly intractable, when considering very large data sets to be processed. Additional tools are needed to automatically produce the huge resulting Directed Acyclic Graphs (DAGs). Even so, DAGs produced can hardly be visualized, especially when considering the complexity of composition operators introduced in section 3. Conversely, workflows of services easily handle the description of input data sets independently from the workflow topology itself. Adding extra inputs does not result in any additional complexity.

2.2. Dynamic data sets

The non-static nature of data description in the service-based approach also enables dynamic extension of the data sets to be processed: a workflow can be defined and executed although the complete input data sets are not known in advance. It will be dynamically fed in as new data are being produced. Indeed, it is common in scientific applications

that data acquisition is an heavy-weight process and that data are being progressively produced. Some workflows may even act on the data production source itself: stopping data production once computations have shown that sufficient inputs are available to produce meaningful results.

Moreover, due to the explicit specification of data in workflows of tasks, it is not possible to define a loop. If there were a loop, a data would depend on itself. Hence, task-based workflows are always Directed and Acyclic Graphs (DAGs). Only in the case where the number of iterations is statically known, a loop may be expressed by unfolding it in the DAG. However, if the loop condition is dynamically determined (*e.g.* in optimization loops that are very frequent in scientific applications), the task-based approach cannot be used. In a workflow of services, there may exist loops in the graph of services since it does not imply a circular dependency on the data. This enables the implementation of more complex control structures.

Some services may need to consider the complete data sets simultaneously to perform computations such as statistical parameters (*e.g.* the mean of all data). These services are synchronization barriers inside the workflow execution. Such constraints can easily be expressed in the service-based framework. The execution engine just needs to know the specific nature of the service for calling it only one time once the complete input data sets have be produced.

Most importantly, the dynamic extensibility of input data sets for each service in a workflow can also be used for defining different data composition strategies as introduced in section 3. The data composition patterns offer a very powerful tool for describing complex data processing scenarios as needed in scientific applications. For the users, this means the ability to describe and schedule very complex processings in an elegant and compact framework.

2.3. Efficient processings of data-intensive workflows

Although very convenient for representing workflows independently from data to be processed, the service-based approach introduces an extra layer between the workflow manager and the execution infrastructure that hides the one from the other [9]. Conversely, the task-based approach does not suffer this limitation. When considering grid infrastructures with a large potential for parallelism and optimization in data-intensive applications, this needs to be taken into account to avoid performance drops. This problem is addressed in section 5 of this paper.

3. Data composition strategies

Each service in a data-intensive workflow of services is receiving input data on its input ports. Depending on the desired service semantic, the user might envisage various input composition patterns between the different input ports.

3.1. Basic data composition patterns

Although not exhaustive, there are two main data composition patterns very frequently encountered in scientific applications that were first introduced in the Taverna workbench [15]. They are illustrated in figure 1. Let $\mathbf{A} = \{\mathrm{A}_0, \mathrm{A}_1, \ldots, \mathrm{A}_n\}$ and $\mathbf{B} = \{\mathrm{B}_0, \mathrm{B}_1, \ldots, \mathrm{B}_m\}$ be two input data sets.

The *one-to-one* composition pattern (left of figure 1) is the most common. It consists in processing two input data sets pairwise in their order of arrival. This is the classical case where an algorithm needs to process every pair of input data independently. An example is a matrix addition operator: the sum of each pair of input matrices is computed and returned as a result. We will denote $\oplus$ the one-to-one composition operator. $\mathbf{A} \oplus \mathbf{B} = \{A_1 \oplus B_1, A_2 \oplus B_2, \ldots\}$ denotes the set of all outputs. For simplification, we will denote $A_1 \oplus B_1$ the result of processing the pair of input data (A_1, B_1) by some service. Usually, the two input data sets have the same size ($m = n$) when using the one-to-one operator, and the cardinality of the results set is $m = n$. If $m \neq n$, a semantics has to be defined. We will consider that only the $\min(m, n)$ first pieces of data are processed in this case.

The *all-to-all* composition pattern (right of figure 1) corresponds to the case where all inputs in one data set need to be processed with all inputs in the other data set. A common example is the case where all pieces of data in the first input set are to be processed with all parameter configurations defined in the second input set. We will denote $\otimes$ the all-to-all composition operator. The cardinality of $\mathbf{A} \otimes \mathbf{B} = \{A_1 \otimes B_1, A_1 \otimes B_2 \ldots A_1 \otimes B_m, A_2 \otimes B_1 \ldots A_2 \otimes B_m \ldots\ldots A_n \otimes B_1 \ldots A_n \otimes B_m\}$ is $m \times n$.

Note that other composition patterns with different semantics could be defined (*e.g. all-to-all-but-one* composition). However, they are more specific and consequently more rarely encountered. Combining the two operators introduced above enable very complex data composition patterns, as will be illustrated below. Adding different operators with a defined semantic would not change significantly the remaining of this paper and we will limit ourselves to the one-to-one and the all-to-all operators.

3.2. Combining data composition patterns

As illustrated at the left of figure 2, the pairwise one-to-one and all-to-all operators can be combined to compose data patterns for services with an arbitrary number of input ports. In this case, the priority of these operators needs to be explicitly provided by the user. We are using parenthesis

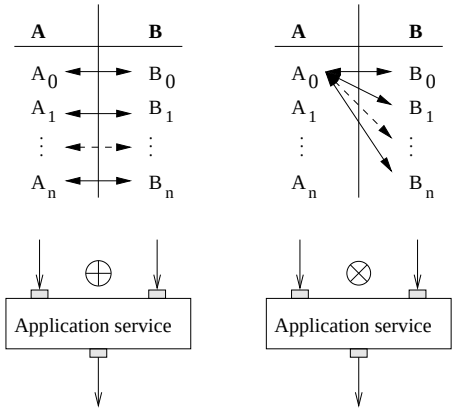

Figure 1. Action of the *one-to-one* (left) and *all-to-all* (right) operators on the input data sets

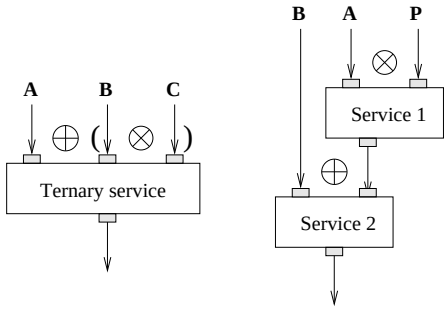

Figure 2. Combining composition operators: multiple input service (left) and cascade of services (right)

in our figures to display priorities explicitly. If the input data sets are $\mathbf{A} = \{A_0, A_1\}$, $\mathbf{B} = \{B_0, B_1\}$, and $\mathbf{C} = \{C_0, C_1, C_2\}$, the following data would be produced in this case:

$$\mathbf{A} \oplus (\mathbf{B} \otimes \mathbf{C}) = \left\{ \begin{array}{ll} A_0 \oplus (B_0 \otimes C_0), & A_1 \oplus (B_1 \otimes C_0), \\ A_0 \oplus (B_0 \otimes C_1), & A_1 \oplus (B_1 \otimes C_1), \\ A_0 \oplus (B_0 \otimes C_2), & A_1 \oplus (B_1 \otimes C_2) \end{array} \right\} \quad (1)$$

Successive services may also use various combinations of data composition operators as illustrated at the right of figure 2. The example given corresponds to a classical situation where an input data set, say two pieces of data $\mathbf{A} = \{A_0, A_1\}$, is processed by a first algorithm (using different parameter configurations, say $\mathbf{P} = \{P_0, P_1, P_2\}$), before being delivered to a second service for processing with a matching number of data, say $\mathbf{B} = \{B_0, B_1\}$. The output data set would be:

$$\mathbf{B} \oplus (\mathbf{A} \otimes \mathbf{P}) = \left\{ \begin{array}{ll} B_0 \oplus (A_0 \otimes P_0), & B_1 \oplus (A_1 \otimes P_0), \\ B_0 \oplus (A_0 \otimes P_1), & B_1 \oplus (A_1 \otimes P_1), \\ B_0 \oplus (A_0 \otimes P_2), & B_1 \oplus (A_1 \otimes P_2) \end{array} \right\} \quad (2)$$

As can be seen, composition operators are a powerful tool for data-intensive application developers who can represent complex data flows in a very compact format. Although the one-to-one operator preserves the input data sets cardinality, the all-to-all operator may leads to drastic increases in the number of data to be processed.

3.3. State of the art in data composition

Taverna [15].
The one-to-one and the all-to-all data composition operators were first introduced and implemented in the Taverna workflow manager. They are part of the underlying Scufl workflow description language. In this context, they are known as the *dot product* and *cross product iteration strategies* respectively. The strategy of Taverna for dealing with input sets of different sizes in a one-to-one composition is to produce the $\min(m, n)$ first results only. However, the semantics adopted by Taverna when dealing with a composition of operators as illustrated in figure 2 is not straight forward. In the ternary service (left of figure 2), Taverna will produce the:

$$\mathbf{A} \oplus_{\text{Taverna}} (\mathbf{B} \otimes \mathbf{C}) = \{ \; A_0 \oplus (B_0 \otimes C_0), \quad A_1 \oplus (B_1 \otimes C_0) \; \} \quad (3)$$

output set. Given that only two input data are available on the first service port, the $\min(m, n)$ truncation rule of the one-to-one (dot product) operator applies. Note that changing the priority of operators will produce a different output. Indeed,

$$(\mathbf{A} \oplus_{\text{Taverna}} \mathbf{B}) \otimes \mathbf{C} = \left\{ \begin{array}{l} \forall i, (A_0 \oplus B_0) \otimes C_i, \\ \forall i, (A_1 \oplus B_1) \otimes C_i \end{array} \right\} \quad (4)$$

Taverna proposes a graphical interface for allowing the user to define the desired priority on the data composition operators.

In the case of the example given in the right of figure 2, the priority on the data composition is implicit in the workflow. There is no user control on it. In this case, Taverna will produce:

$$\mathbf{B} \oplus_{\text{Taverna}} (\mathbf{A} \otimes \mathbf{P}) = \{ \; B_0 \oplus (A_0 \otimes P_0), \quad B_1 \oplus (A_1 \otimes P_0) \; \} \quad (5)$$

More data will be produced at the output of the Service1 (namely, $A_0 \otimes P_1, A_1 \otimes P_1, A_0 \otimes P_2, A_1 \otimes P_2$) but the truncation semantics of the one-to-one operator will apply in the second service and only two output data will be produced. Note that this semantics differs from the one that we consider and that is illustrated in equation 2.

Kepler [11] and Triana [17].
The Kepler and the Triana workflow managers only implement the one-to-one composition operator. This operator is

implicit for all data composition inside the workflow and it cannot be explicitly specified by the user.

We could implement an all-to-all strategy in Kepler by defining specific actors but this is far from being straight forward. Kepler actors are blocking when reading on empty input ports. The case where two different input data sets have a different size (common in the all-to-all composition operator) is not really taken into account. Similar work can be achieved in Triana using the various *data stream* tools provided. However, in both cases, the all-to-all semantics is not handled at the level of the workflow engine. It needs to be implemented inside the application workflow.

MOTEUR.

We designed the MOTEUR workflow engine so that it implements the semantics of the operators defined in this paper. MOTEUR recognizes both one-to-one and all-to-all operators (it does recognize Scufl workflows) but it uses the algorithm introduced in section 4 to define the combination semantics.

4. Data composition algorithm

Even considering simple examples such as the ones shown in figure 2, the semantics of combining data composition operators is not straight forward. Different workflow engines have different capabilities and implement different combination strategies. Our goal is to define a clear and intuitive semantics for such combinations. We propose an algorithm to implement this data combination strategy.

Even though Taverna provides the most advanced data composition techniques, the semantics described in equation 5 is not intuitive. Given that two correlated input data sets **A** and **B**, with the same size, are provided, the user can expect that the data A_i will always be analyzed with the correlated data B_i, regardless of the algorithm parameters P_j considered. We therefore adopt the semantics proposed in equation 2 where A_i is consistently combined with B_i.

To formalize and generalize this approach, we need to consider the complete data flows to be processed in the application workflow. In the reminder of this paper, we will consider the very general case, common in scientific applications, where the user needs to independently process sets of input data $\mathbf{A}, \mathbf{B}, \mathbf{C} \ldots$ that are divided into *data groups*. A group is a set of input data tuples that defines a relation between data coming from different sets. For instance:

$$\begin{array}{l} \mathrm{G} = \{(A_0, B_0, C_0), (A_1, B_1, C_1), (A_2, B_2, C_2)\} \\ \mathrm{H} = \{(A_4, B_0), (A_1, B_2), (A_2, B_5), (A_6, B_6)\} \end{array} \quad (6)$$

are two groups establishing a relation between 3 data triplets and 4 data pairs respectively. The relations between input data depend on the application and can only be specified by the user. However, we will see that this definition can be explicit (as illustrated above) or implicit, just considering the workflow topology and the order in which input data are received by the workflow manager.

4.1. Data composition operator semantics

We consider that the one-to-one composition operator does only make sense when processing related data. Therefore, only data connected by a group should be considered for processing by any service. When considering a service directly connected to input data sets, determining relations between data is straight forward. However, when considering a complete application workflow such as the one illustrated in figure 3, other services (*e.g.* S_4) need to determine which of their input data are correlated. The one-to-one composition operator does introduce the need for the algorithm described below.

Conversely, the all-to-all operator does not rely on any pre-determined relation between input data. Any number of inputs can be combined, with very different meaning (such as data to process and algorithm parameters). Each data received as input yields to one or more invocations of the service for processing.

4.2. Combination semantics

The left of figure 3 represents a sample workflow made of 4 application services and combining the one-to-one and the all-to-all composition operators. In the center of the figure is represented the directed graph of the data sets produced. Given 4 input data sets, $\mathbf{A}, \mathbf{B}, \mathbf{P}$ and $\mathbf{Q}$, the complete workflows produces

$$((\mathbf{A} \oplus \mathbf{B}) \otimes \mathbf{P}) \oplus ((\mathbf{A} \oplus \mathbf{B}) \otimes \mathbf{Q}). \quad (7)$$

as output of the S_4 service. Given the one-to-one operator semantics described above, the data set $\mathbf{A} \oplus \mathbf{B}$ produced by the first service will be non empty if and only if data in $\mathbf{A}$ and $\mathbf{B}$ are related through a group G that is represented at the top of the figure (A_i, the i^{th} element of $\mathbf{A}$, is correlated with B_i, the i^{th} element of $\mathbf{B}$).

Considering the service S_4, it is not trivial to determine the content of the output data set, resulting from a one-to-one composition of the two inputs $(\mathbf{A} \oplus \mathbf{B}) \otimes \mathbf{P}$ and $(\mathbf{A} \oplus \mathbf{B}) \otimes \mathbf{Q}$. Intuitively, two input data $(A_i \oplus B_i) \otimes P_k$ and $(A_j \oplus B_j) \otimes Q_l$ should be combined only if $i = j$. Indeed, combining A_i with B_i, or a subsequent processing of these data, does make sense given that the user established a relation between this input pair through the group G. Conversely, there is no relation between $A_i \oplus B_i$ on the one side and any P_k or Q_l that are combined in an all-to-all operation on the other side. Therefore, the processing of $((A_i \oplus B_i) \otimes P_k) \oplus ((A_i \oplus B_i) \otimes Q_l)$ does make sense for all k and all l.

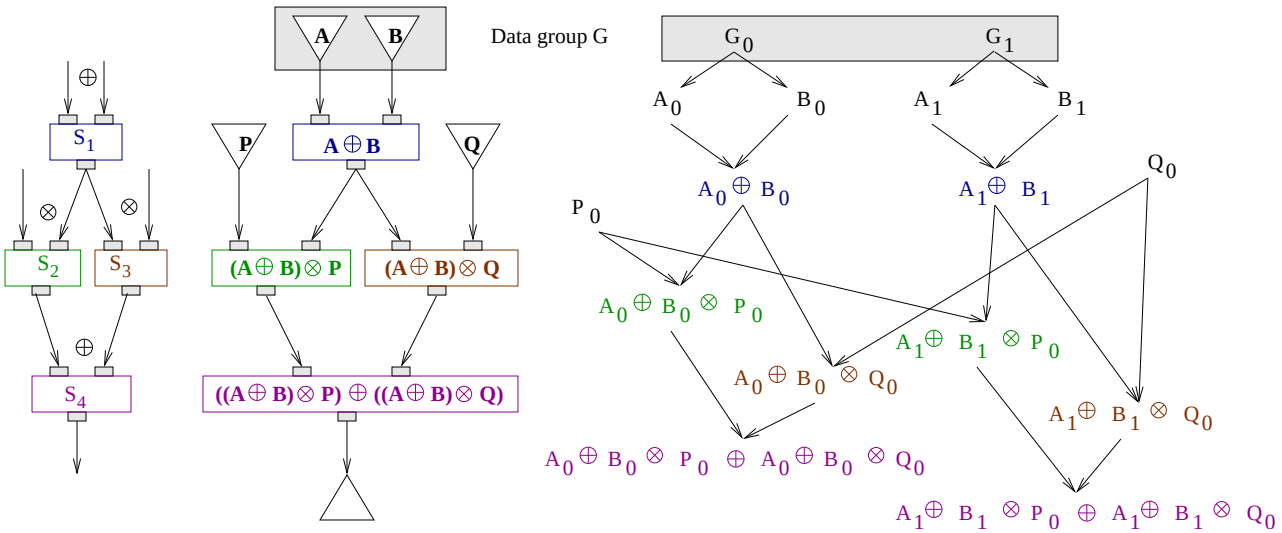

Figure 3. Workflow example (left), associated data sets directed graph (center), and part of the associated directed acyclic data graph.

To formalize this approach we need to consider the data production Directed Acyclic Graph that is represented in right of figure 3. This graph shows how all pieces of input are combined by the different processings. At the roots of the graph, the *input* data are parents of all *produced* data. The formal relation between each data pair (A_i, B_i) is represented through a group instantiation G_i, parent of both A_i and B_i. We will name *orphan* data, input data that have no group parent such as P_0 and Q_0. The directed data graph is constructed from the roots (workflow inputs) to the leafs (workflow outputs) by applying the two following simple rules implementing the semantics of the one-to-one and the all-to-all operators respectively:

1. Two data are always combined in an all-to-all operation.
2. Two data (graph nodes) are combined in a one-to-one operation **if and only if** there exists a common ancestor to both data in the data graph.

The interpretation of the first rule is straight forward. The second rule is illustrated by the full data graph displayed at the right of figure 3. For instance, the data $A_0 \oplus B_0$ is produced from A_0 and B_0 because there exists a common ancestor G_0 to both A_0 and B_0. Similarly, $((A_0 \oplus B_0) \otimes P_0) \oplus ((A_0 \oplus B_0) \otimes Q_0)$ is computed because $A_0 \oplus B_0$ is a common ancestor to $(A_0 \oplus B_0) \otimes P_0$ and $(A_0 \oplus B_0) \otimes Q_0$. There exists other common ancestors such as A_0, B_0, and G_0 but it is not needed to go back further in the data graph as soon as one of them has been found. Note that in a more complex workflow topology, the common ancestor does not need to be an immediate parent. It can be easily demonstrated by recurrence that following this rule, two input data sets may be composed one-to-one if and only if there exists a grouping relation between them at the root of the data graph.

4.3 Algorithm and implementation

To implement the data composition operators semantic introduced above, MOTEUR dynamically resolves the data combination problem by applying the following algorithm:

1. Build the directed graph of the data sets to be processed.
2. Add data groups to this graph.
3. Initialize the directed acyclic data graph:
   (a) Create root nodes for each group instance G_i and add a child node for each related data.
   (b) Create root nodes for each orphan data.
4. Start the execution of the workflow.
5. For each tuple of data to be processed:
   (a) Update the data graph by applying the two rules corresponding to the one-to-one and the all-to-all operators.
   (b) Loop until there are no more data available for processing in the workflow graph.

To implement this strategy, MOTEUR needs to keep representations of:

- the topology of the services workflow;

- the graph of data;
- the list of input data that have been processed by each service.

Indeed, the data graph is dynamically updated during the execution. When a new data is produced, its combination with all previously produced data is studied. In particular in an all-to-all composition pattern, a new input data needs to be combined with all previously computed data. It potentially trigger several services invocation. The history of previous computations is thus needed to determine the exhaustive list of data to produce.

The graphs of data also ensures a full traceability of the data processed by the workflow manager: for each data node, the parents and children of the data can be determined. Besides, it provides a mean to unambiguously identify each data produced. This becomes mandatory when considering parallel execution of the workflow introduced in section 5.

4.4. Implicit combinations

The algorithm proposed aims at providing a strict semantics to the combination of data composition operators, while providing intuitive data manipulation for the users. Data groups have been introduced to clarify the semantics of the one-to-one operator. However, it is very common that users are writing workflows without explicitly specifying pairwise relations between the data. The order in which data are declared or send to the workflow inputs are rather used as an implicit relation.

To ease the workflow generation from the user point of view, groups can be implicitly generated when they are not explicitly specified by the user. Figure 4 illustrates two different cases. On the left side, the reason for generating an implicit group is straight forward: two input data sets are being processed through a one-to-one service. But there may be more indirect cases such as the one illustrated on the right side of the figure. The systematic rule that can be applied is to create an implicit group for each *one-to-one* operator whose input data are orphans. For example, in the case illustrated in left of figure 4, the input data sets $\mathbf{A}$ and $\mathbf{B}$ are orphans and bound *one-to-one* by the S_1 service. An implicit group is therefore created between $\mathbf{A}$ and $\mathbf{B}$. In the case illustrated in the right side of figure 4, the implicit group will be created between the two inputs of service S_2. There will therefore be an implicit grouping relation between each output of the first service $S_1(A_i)$ and B_i.

The implicit groups are created statically by analyzing the workflow topology and the input data sets before starting the execution of the workflow.

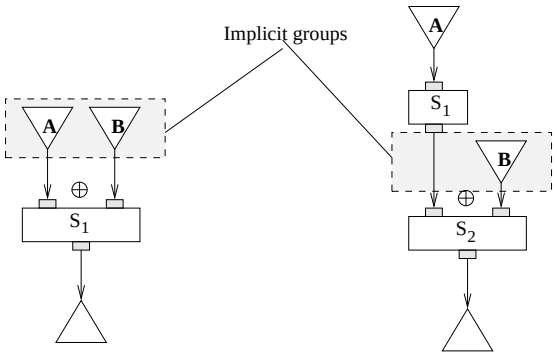

Figure 4. Implicit groups definition.

4.5. Coping with data fragments

So far, we have only considered the case were the number of outputs of a service matches the number of inputs. In some cases though, an application service will split input data in smaller fragments, either for dealing with smaller data sets (*e.g.* a 3D medical image is split in a stack of 2D slices) or because the service code function implies that it produces several outputs for each input. The workflow displayed in figure 5 illustrates such a situation. The service S_1 is splitting each input data (*e.g.* A_0) in several fragments (A_0^0, A_0^1 and A_0^2).

In the example given in figure 5, it is expected that service S_2 will receive the same number of data on both input ports (one-to-one composition operator). However, there is no way for the user to specify an explicit grouping between two data sets. Grouping the data sets $\mathbf{A}$ with $\mathbf{B}$ would only create a relation between A_0 and B_0. Therefore, the fragments A_0^0, A_0^1 and A_0^2, children of A_0, would all be related to B_0 and the service S_2 would produce

$$\mathbf{A} \oplus \mathbf{B} = \{A_0^0 \oplus B_0, A_0^1 \oplus B_0, A_0^2 \oplus B_0\}. \quad (8)$$

Instead, the implicit grouping strategy will group $S_1(A_0)$ outputs with $\mathbf{B}$. Consequently, the grouping will result in the data graph shown in right of figure 5 and the output produced will be

$$\mathbf{A} \oplus \mathbf{B} = \{A_0^0 \oplus B_0, A_0^1 \oplus B_1, A_0^2 \oplus B_2\} \quad (9)$$

as expected. Note that the number of inputs to service S_2 needs to be consistent in this case.

5. Exploiting grid parallelism

The main interest for using grid infrastructures in the processing of data-intensive applications is to exploit the potential application parallelism thanks to the distributed

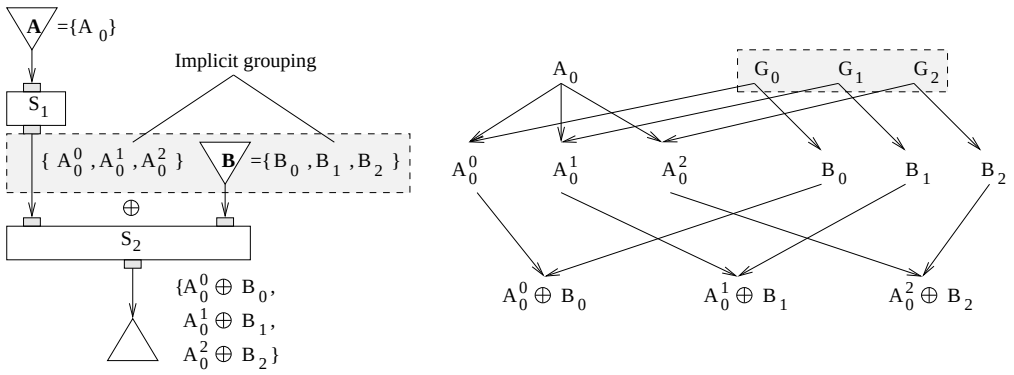

Figure 5. Implicit groups relating data fragments (A_0^0, A_0^1, A_0^2) and input B.

grid resources available. There are three different levels of parallelism that can be exploited when considering any application workflow. The parallel processing of a workflow requires to cautiously handle the data flows to avoid causality errors that would lead to incorrect computations.

5.1. Three parallelism levels

Workflow parallelism. The first level of parallelism that can be exploited is the intrinsic workflow parallelism depending on the graph topology. For instance if we consider the simple example presented in figure 3, services S_2 and S_3 can be executed in parallel.

Data parallelism. Data are processed independently from each other. Therefore, different input data can be processed in parallel on different resources. This optimization may lead to considerable performance improvements given the high level of parallelism achievable in data-intensive applications.

Services parallelism. The processing of two different data sets by two different services are totally independent. This pipelining model, very successfully exploited inside CPUs, can be adapted to sequential parts of service-based workflows. Considering the simple workflow represented on right side of figure 4 for example, $S_1(A_1)$ first needs to be computed before S_2 may be invoked. Once this result is available, $S_2(A_1, B_1)$ may be computed in parallel of $S_1(A_2)$. In theory, service parallelism brings an additional level of performance improvement beyond data parallelism only if all data could not be processed in parallel at the same time by lack of resources or if the time for processing different data is not constant. In practice this is always the case on production grid infrastructures as the number of resources is bounded and the load of the infrastructure leads to variable execution times.

5.2. State of the art in service-based workflow managers

Workflow parallelism is usually implemented in existing workflow managers.

Taverna implements data parallelism (known as *multiple threads* in this context). However, data parallelism is limited to a fixed number of threads specified in the Scufl workflow description language. It cannot dynamically adapt to the size of data sets to be processed. Service parallelism is not supported yet but this feature has been proposed for the next major release of the engine (version 2).

Kepler implements services parallelism through the *Physical Network* (PN) director. There is no data parallelism in Kepler.

MOTEUR was designed to optimize the performance of data-intensive applications on grids by implementing the three level of parallelism. To our knowledge, this is the first service-based workflow engine doing so.

5.3. Parallel processings and data flows

Executing a workflow with data and service parallelism has non trivial consequences on the application processing and data flows. MOTEUR is making concurrent calls to grid submission services wrapping application codes using the Web Services or GridRPC interfaces. Although Web Services are stateless, MOTEUR is holding the state of each service invocation. It is handling the concurrency without requiring the use of stateful services such as defined in the WSRF framework.

Data parallelism leads to a disordering of the data flows and causality problems as a data may overtake another one during the workflow processing (depending on the computation time required and the grid resources used for processing [9]). To properly tackle this problem, computed data have to be traced during the execution. The data graph built

by MOTEUR unambiguously identifies each processed data through its processing history. This ensures that data are correctly combined, ignoring their order of arrival and only considering their provenance.

Service parallelism implies that data are progressively delivered to all workflow services as they are being produced by their parents. The complete input data set for each service is not known until the end of the workflow execution. Therefore, MOTEUR needs to cache and dynamically update the data already received for processing by each service in order to decide of the data compositions to be scheduled when a new piece of data is delivered to a service. If the new data matches the composition operator rule, it triggers one or more service invocations.

6. Concrete application example

We are using MOTEUR to develop applications to medical image analysis. 3D medical images are large (commonly in the order of tens to hundreds of megabytes per image) and many medical imaging application require processing of full databases [12]. A good example is the *bronze standard* application [9] which aims at assessing medical image registration algorithms.

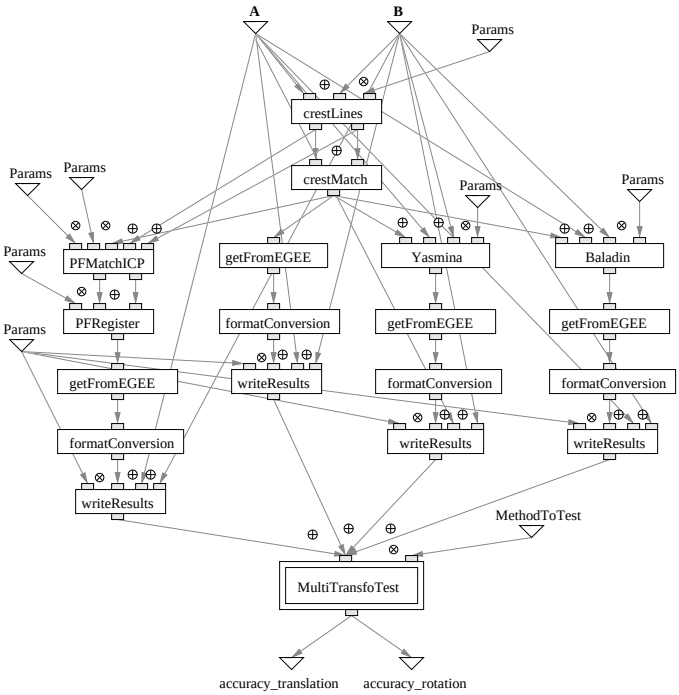

Figure 6. Application workflow

Figure 6 displays the bronze standards application workflow. Each one of the 19 application services involved is represented by a box and data flows are represented by arrows. The bottom service (*MultiTransfoTest*) is surrounded by a double square to indicate that it corresponds to a data synchronization barrier (see section 2.2). It can be seen that both data composition operators are used intensively.

This application is using pairs of input image lists (inputs **A** and **B**). We made runs involving images acquired from 1, 7 and 25 patients leading to the processing of 6 to 126 images as reported in figure 7. The execution results reported show two kinds of execution: a reference execution (red curve) where only intrinsic workflow parallelism is exploited (as available in most service-based workflow engines) and a fully parallel execution (blue curve) exploiting all levels of parallelism implemented in MOTEUR (see section 5). Both executions were using the EGEE [4] production grid infrastructure. EGEE is providing thousands of CPUs distributed all over Europe and beyond for parallel processing of applications. It can be seen that MOTEUR provides very good performances with a speed-up in the order of 10 when considering a full run (126 images) as compared to a reference execution on the same grid infrastructure.

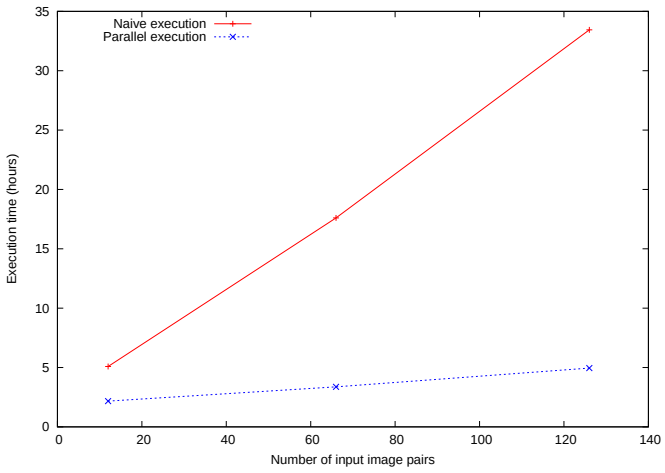

Figure 7. Execution performances

Note that the execution on 1 to 25 patients (6 to 126 images) of the full workflow leads to the execution of 72 to 756 computing tasks on the grid infrastructure. In a task-based approached, the user would need to draw graphs made of 72 to 756 tasks to represent such executions. In our framework, the application workflow representation remains relatively simple (19 services) and it is independent on the size of the input data sets.

7. Conclusions

Workflow description languages and execution engines ease the development of scientific applications as they provide a high level representation to describe the application logic. In addition, data composition operators provide a powerful, compact and intuitive mean for describing complex data flows. Parallel execution of these data-intensive workflows provide a support for transparently accessing distributed grid resources and improving the application performances.

In this paper we define a precise and coherent semantics for data composition operators and we introduced an algorithm that implements it in the workflow execution engine. The algorithm is based on the construction of a data directed acyclic graph that is also used to overcome the causality problems arising during parallel execution.

We have shown how implicit grouping of data inputs enables transparent implementation of the operators semantics when the user is designing a workflow. This grouping strategy also coherently enables data splitting.

Our algorithm has been implemented in the MOTEUR data-intensive workflow engine. MOTEUR is freely available under CeCILL (a GPL-like) license[4].

8. Acknowledgment

This work is partially funded by the French research program “ACI-Masse de données” (http://acimd.labri.fr/), AGIR project (http://www.aci-agir.org/). We are grateful to the EGEE IST European project for providing the grid infrastructure and user support.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services. Technical Report version 1.1, `http://www-128.ibm.com/developerworks/library/ws-bpel/`, May 2003.

[2] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task Scheduling Strategies for Workflow-based Applications in Grids. In *CCGrid*, Cardiff, UK, 2005.

[3] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, and G. Mehta et al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):9 – 23, 2003.

[4] European IST project of the FP6, Enabling Grids for E-sciencE, apr. 2004-mar. 2006. `http://www.eu-egee.org`.

[5] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *International Conference on Network and Parallel Computing (IFIP)*, volume 3779, pages 2–13. Springer-Verlag LNCS, 2005.

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.

[7] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying and automating data derivation. In *Scientific and Statistical DB Management*, Edinburgh, Scotland, 2002.

[8] T. Glatard, J. Montagnat, and X. Pennec. An optimized workflow enactor for data-intensive grid applications. Technical Report 05.32, I3S Laboratory, Sophia Antipolis, France, `http://www.i3s.unice.fr/ glatard/publis/RR-05.32-T.GLATARD.pdf`, Oct. 2005.

[9] T. Glatard, J. Montagnat, and X. Pennec. Grid-enabled workflows for data intensive applications. In *Computer Based Medical Systems (CBMS'05), special track on Grids for Biomedicine and Bioinformatics*, Dublin, Ireland, June 2005.

[10] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, Feb. 2003.

[11] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, 2005.

[12] J. Montagnat, V. Breton, and I. Magnin. Using grid technologies to face medical image analysis challenges. In *Biogrid'03, proceedings of the IEEE CCGrid03*, Tokyo, Japan, May 2003.

[13] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Technical report, Global Grid Forum (GGF), jul 2005.

[14] National Research Grid Initiative (NAREGI). `http://www.naregi.org`.

[15] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20):3045–3054, 2004.

[16] Open Science Grid (OSG). `http://www.opensciencegrid.org`.

[17] I. Taylor, I. Wand, M. Shields, and S. Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice & Experience*, 17(1–18), 2005.

[18] W. M. Van Der Aalst and A. H. Ter Hofstede. YAWL: Yet Another Workflow Language. Technical Report FIT-TR-2002-06, Queensland University of Technology, Brisbane, Australia, 2002.

[19] W. World Wide Web Consortium. Web Services Description Language (WSDL) 1.1, mar 2001.

[20] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD Record*, 34(3):44–49, Sept. 2005.

[4] MOTEUR, `http://www.i3s.unice.fr/~glatard/software.html`